# Monotonic Queue

Bruce Nan

# Queue Review

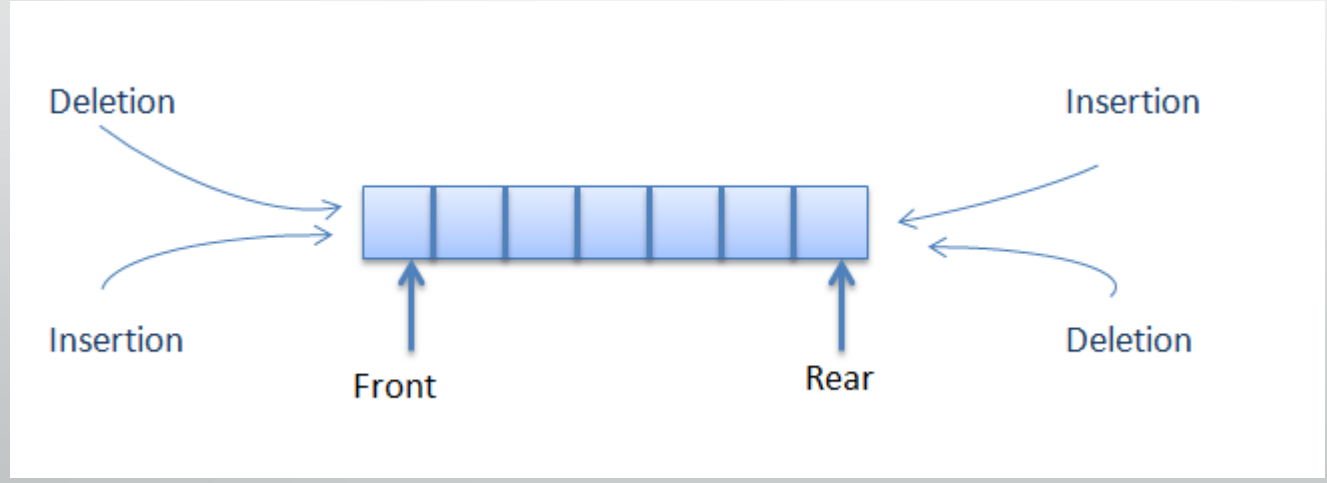
- Queue is a linear data structure.
- Insert at the rear side and delete at the front side
- Data going into the queue first, leaves out first.
- Queue is also known as FIFO data structure (**F**irst-**I**n, **F**irst-**O**ut).

# Basic Queue Operations

- Enqueue – Adds an item to the rear side of a queue
- Deque – Removes an item from the front side of a queue
- C++: push , front, pop
- Java : add, peek, poll

# Deque

- A deque is a double-ended queue
- Insertions *and* deletions can occur at *either* end
- Implementation is similar to that for queues



# Monotonic Queue

- **Monotonic queue** is actually a deque
- It just uses some greedy logic to keep the elements in the deque orderly (monotone increasing or monotone decreasing) after each new element putting into the deque.

# Example: Sliding Window Maximum

# Problem Description

- Leetcode: sliding window maximum
- Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

**Example:**

**Input:** *nums* = [1,3,-1,-3,5,3,6,7], and *k* = 3
**Output:** [3,3,5,5,6,7]
**Explanation:**

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

# Brute Force

- For each start position, scan every K elements and find out the maximum
- Time complexity O(N*K)

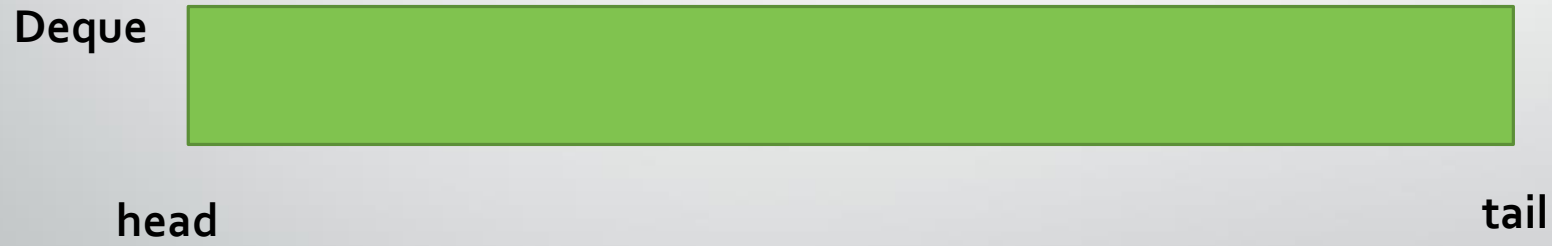
- How to optimize it?

# Idea

- Suppose A[l] and A[r] are in the current sliding window, A[l] ≤ A[r] and l < r. Once A[r] enters the sliding window, we no longer need A[l]
- Why? As the sliding window moves to the right, A[r] will always stay within it longer than A[l].
- Maintain a queue as the sliding window moves. When a new element A[i] enters the sliding window, we pop up any element in the queue which is smaller than A[i]. This procedure ensures the queue elements are in decreasing order. That is why we call it "monotonic queue"
- **Note: the elements beyond its left boundary should be also removed** from the monotonic queue. Thus, we should use a deque.

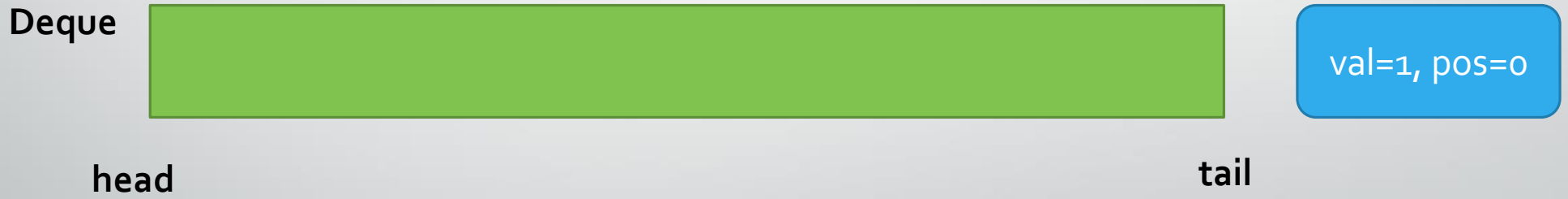
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | | | | | | |



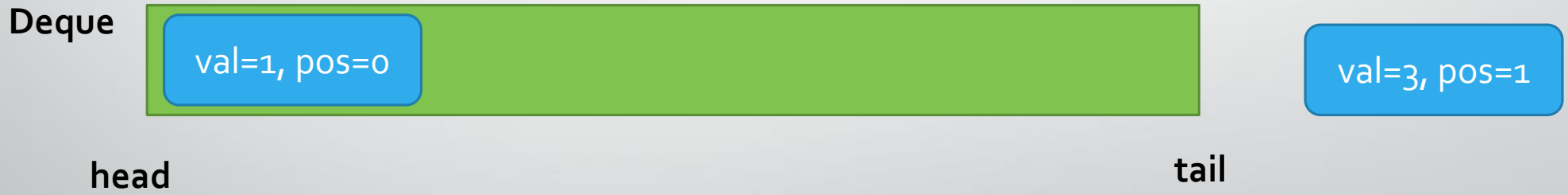
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | | | | | | |



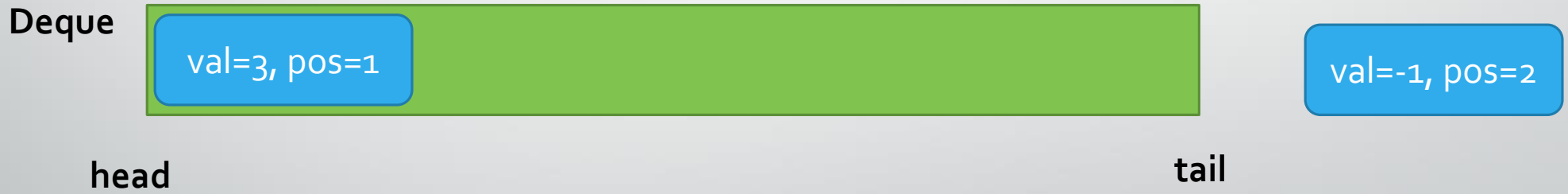
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | | | | | | |



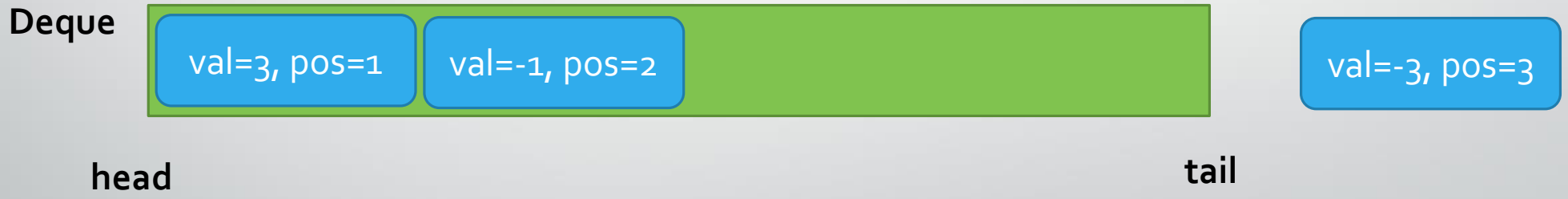
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | | | | | |



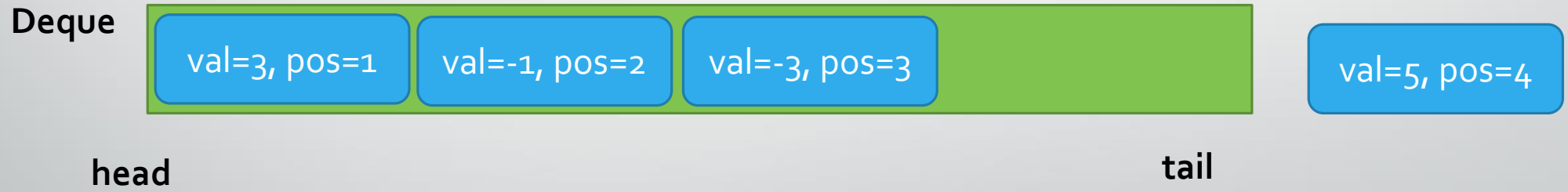
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | 3 | | | | |



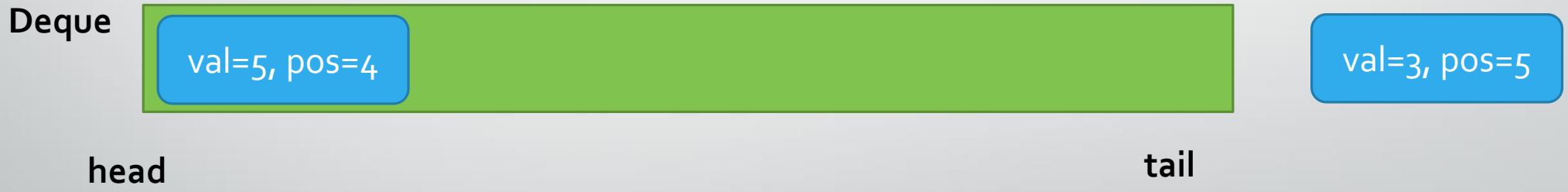
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | 3 | 5 | | | |



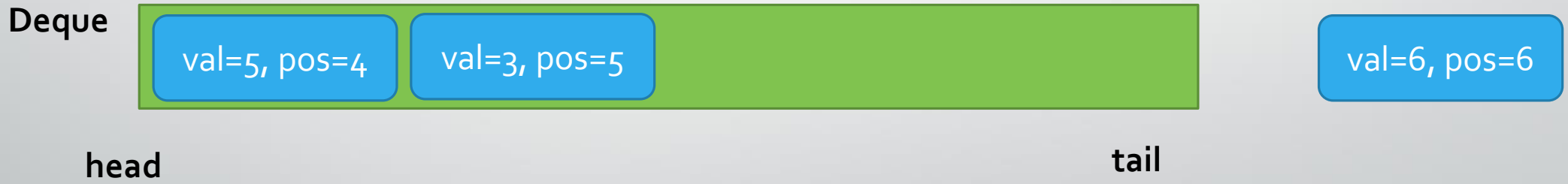
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | 3 | 5 | 5 | | |



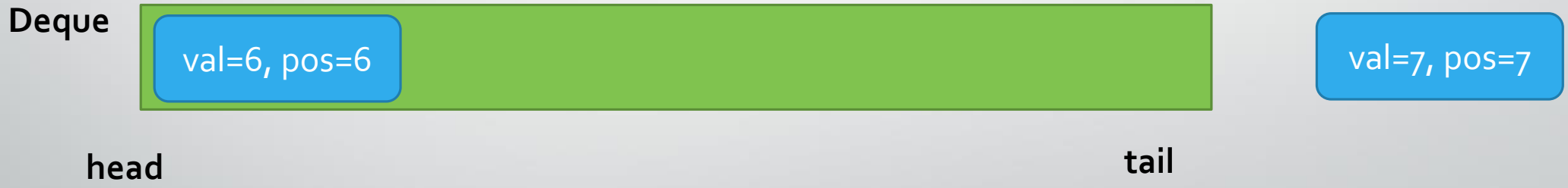
# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | 3 | 5 | 5 | 6 | |



# Monotonic Queue

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 3 | -1 | -3 | 5 | 3 | 6 | 7 |
| Ans | | | 3 | 3 | 5 | 5 | 6 | 7 |



```
for(int i=0; i<n; i++){
    while(!dq.empty() && dq.front().pos <= i-k) dq.pop_front();
    while(!dq.empty() && dq.back().val < a[i]) dq.pop_back();
    dq.push_back({a[i], i});
    if(i >= k-1) ans.push_back(dq.front().val);
}
```

- Each integer is pushed into deque once and popped from deque at most once
- Time complexity O(n)
- Space complexity O(n)

# Example: Subarrays with range <= k

# Problem Description

- DMOPC '15 Contest 6 P5 - A Classic Problem

Given an array with $N$ elements, find the number of subarrays $S$ such that $\max(S) - \min(S) \leq K$

Input:
5 2
0 3 2 1 4

Output:
8

There are 8 subarrays with range <= 2.
[0], [3], [2], [3, 2], [1], [2, 1], [3, 2, 1], [4]

# Brute Force

- There are n^2 subarrays in total. Loop through each subarray to find the max and min.
- Time complexity O(n^2)

# Idea

- Find out the max and min element in  a sliding window [L, R]
- For the current position at R, if the range in [L, R] is <= k, there will be R – L + 1 more subarrays
- Use two monotonic queue to efficiently find out the max and min values in a sliding window

# Monotonic Queue

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 0 | 3 | 2 | 1 | 4 |

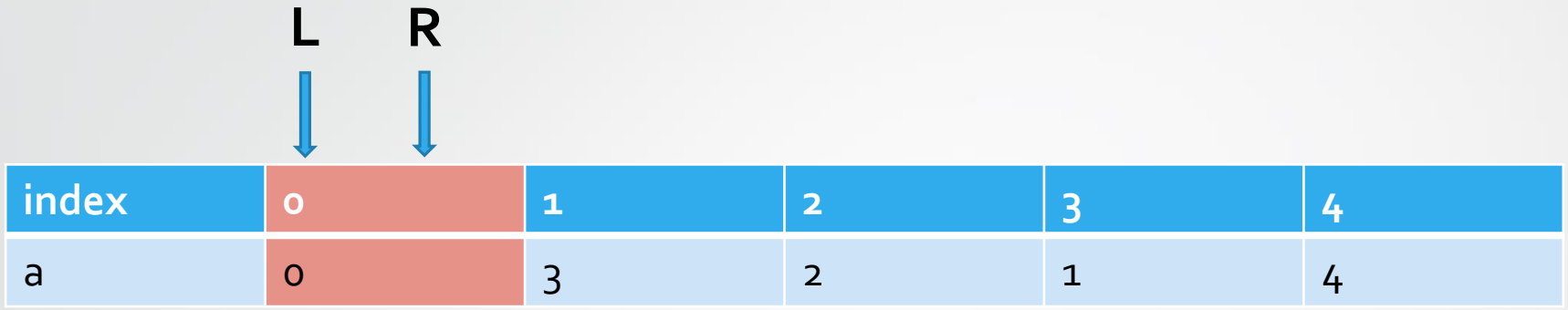
**Ans** = 0

**MaxQ**

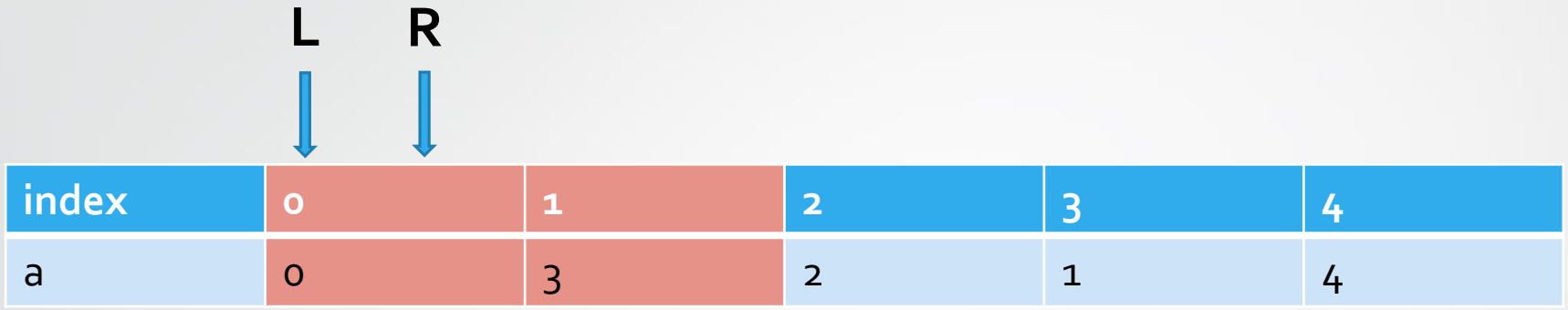
**MinQ**

**head**

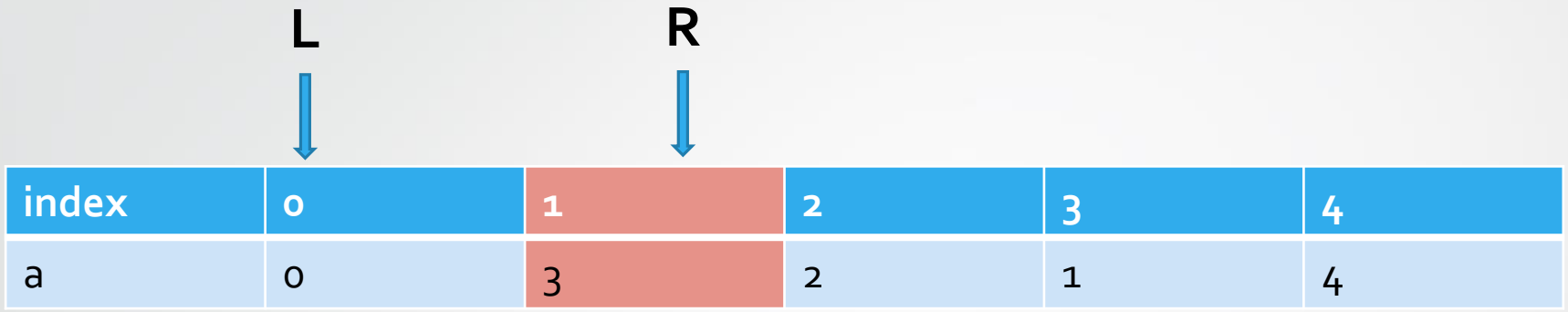
**tail**



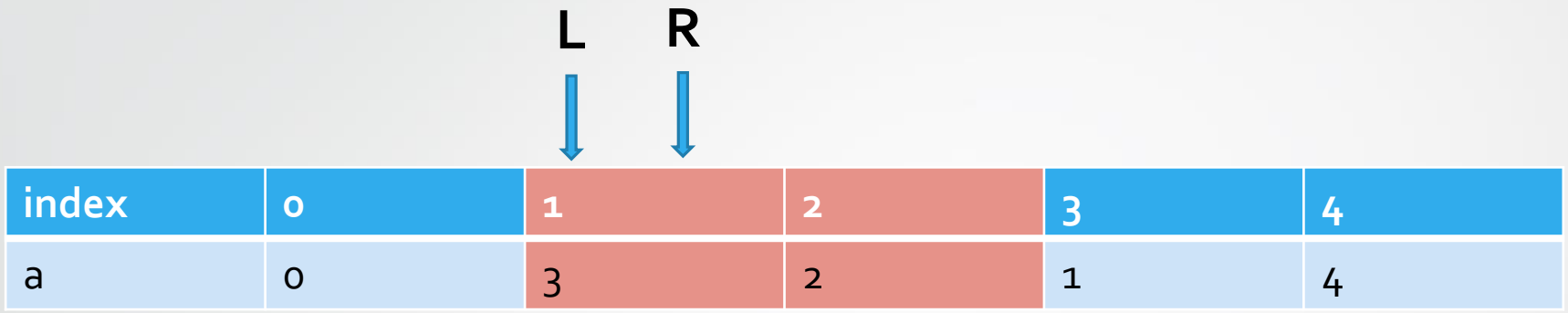

**Ans** = 0 + 1




**Ans** = 0 + 1



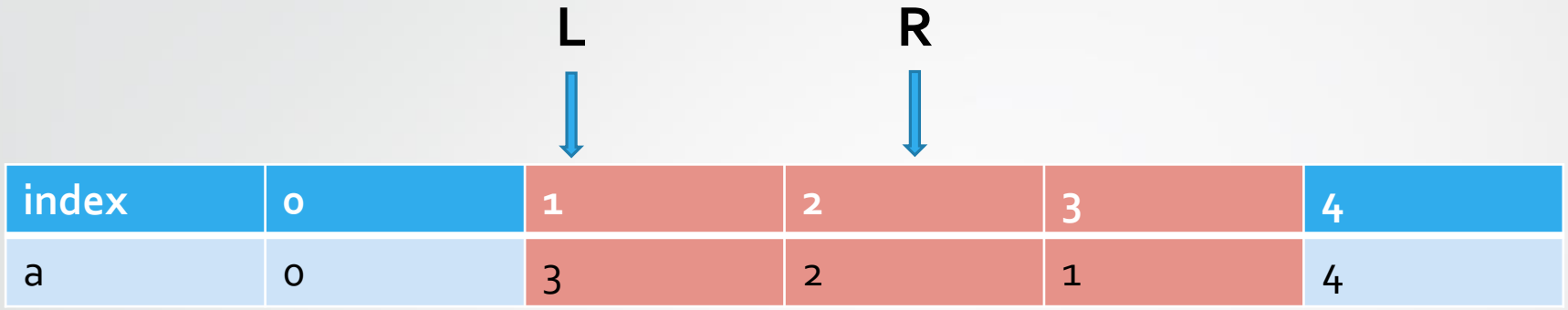

**Ans** = 0 + 1 + 1




**Ans** = 0 + 1 + 1 + 2


head

tail




**Ans** = 0 + 1 + 1 + 2 + 3

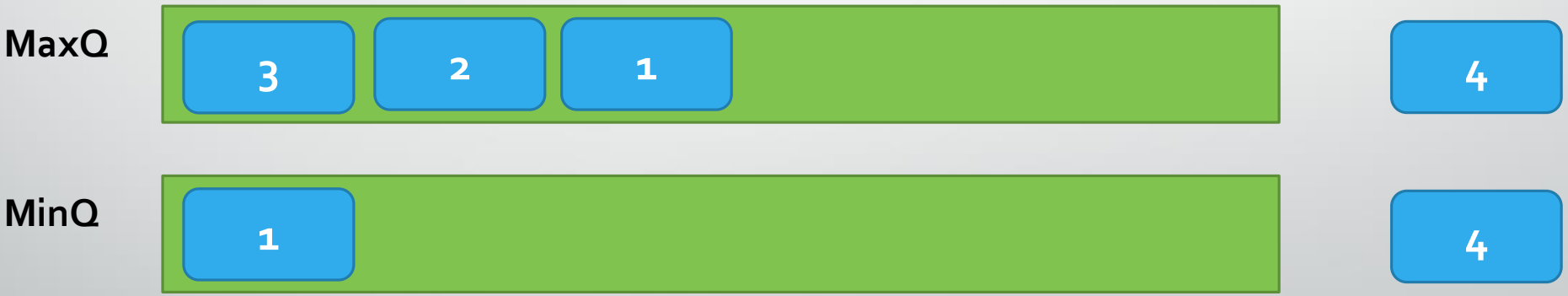

head

tail

L R

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 0 | 3 | 2 | 1 | 4 |

**Ans** = 0 + 1 + 1 + 2 + 3



L

R

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 0 | 3 | 2 | 1 | 4 |

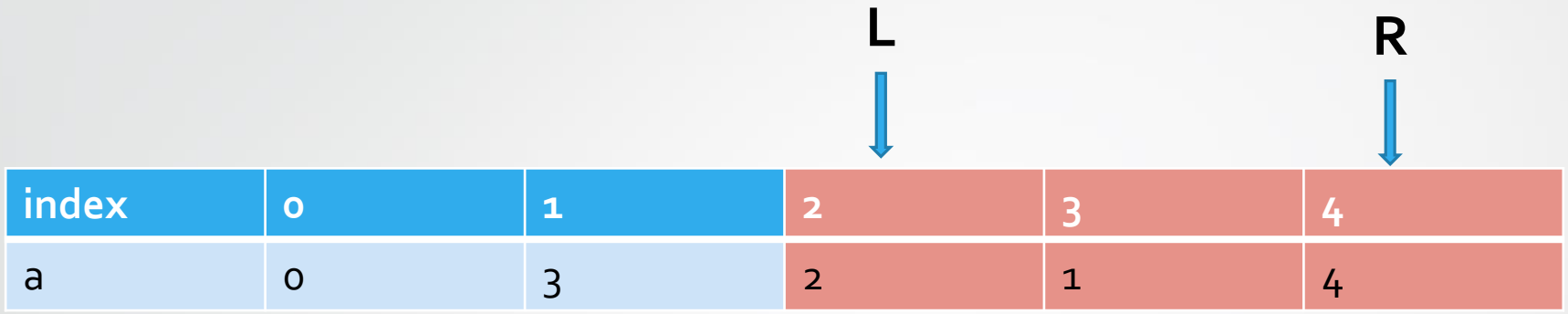
**Ans** = 0 + 1 + 1 + 2 + 3

**MaxQ**

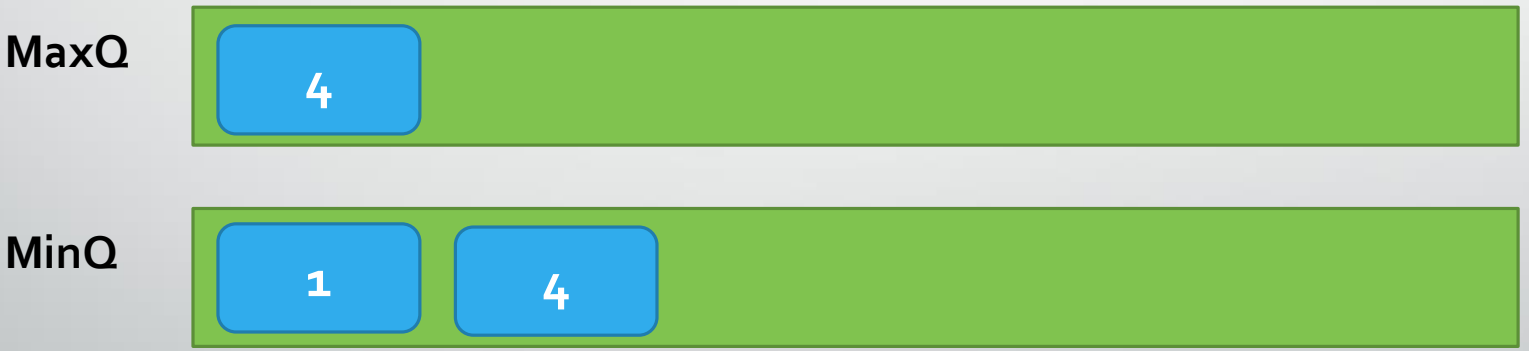

**MinQ**


**head**

**tail**




**Ans** = 0 + 1 + 1 + 2 + 3



L R

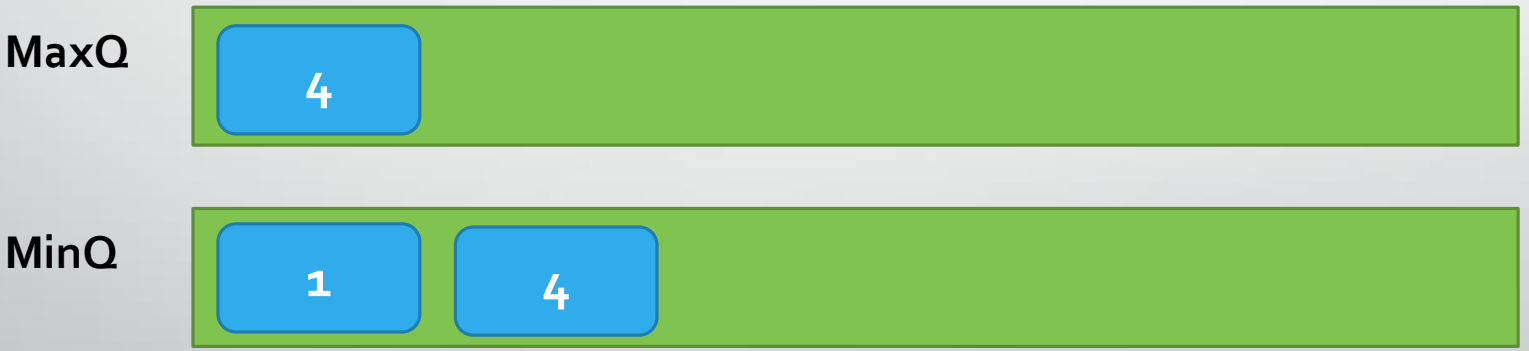
| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 0 | 3 | 2 | 1 | 4 |

**Ans** = 0 + 1 + 1 + 2 + 3

L R

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 0 | 3 | 2 | 1 | 4 |

**Ans** = 0 + 1 + 1 + 2 + 3 + 1 = **8**



```
Deque<Integer> mx = new ArrayDeque(), mi = new ArrayDeque();
for(int i = 0, j = 0; j<N; j++) {
    while( !mx.isEmpty() && mx.getLast() < a[j] ) mx.removeLast();
    while( !mi.isEmpty() && mi.getLast() > a[j] ) mi.removeLast();
    mx.addLast(a[j]);  mi.addLast(a[j]);
    while( mx.getFirst() - mi.getFirst() > K ) {
        if( mx.getFirst() == a[i] ) mx.removeFirst();
        if( mi.getFirst() == a[i] ) mi.removeFirst();
        i++;
    }
    ans += j - i + 1;
}
```

- Each integer is pushed into each deque once and popped from the deque at most once
- Time complexity O(n)
- Space complexity O(n)

# Summary

- Monotonic queue is just a deque
- The elements in deque are ordered (increasing or decreasing order)
- Can efficiently find out the minimal or maximal element in a window
- Each element pushed into deque or popped from deque once, the total time complexity is O(n)

# Practice problems

- DMOPC '15 Contest 6 P5 - A Classic Problem
- DMPG '16 S5 - Pizza Bag
- Baltic OI '07 P3 - Sound